Chapter 8

Storage Classes and Allocation

The VAX C language defines a number of storage-class keywords that specify the scope of an identifier, the location of storage, and the lifetime of the storage allocation. Storage-class modifiers are keywords that you can use with the storage-class and data-type keywords that restrict access to variables. The order of the storage-class keyword, the storage-class modifier, the data-type modifier, and the data-type keyword within the variable declaration does not matter. Each declaration, by virtue of its position in the program source code, has a default storage class, but you may override the default by specifying a storage-class specifier or a storage-class modifier.

This chapter describes the following material:

- The scope of an identifier
- The location of storage
- The lifetime of storage allocation
- The internal storage class
- The static storage class
- The external storage class
- The global storage class
- The data-type modifiers
- The storage-class modifiers

8.1 Scope

The scope of an identifier is that portion of the program in which the identifier has meaning. An identifier has meaning if it is recognized by the compiler, or by the linker. The following sections explain the rules to follow so that your program identifiers will have meaning, to both the compiler and the linker, in all desired portions of your program.

All tags are subject to the same scope rules as other identifiers. A member of a structure or union may have the same name as a member of another structure or union; the scope of the member names can exist concurrently. However, when referencing one of the members in a section of the program where the scopes of both members are concurrent, specify to which structure or union the member belongs. For more information, see Chapter 7.

8.1.1 The Compilation and Linking Process

To understand scope, you must know the VAX C/ULTRIX definitions of function, compilation unit, object file, object module, and program.

When you write VAX C source programs, you can use several methods to compile a program. You can compile a single source file, or a group of source files, into a single object file. The group of source file(s) compiled to create a single object file is called the compilation unit. When documentation to other implementations refers to the source file, the VAX C/ULTRIX equivalent is the compilation unit, not necessarily a single source file. The single, resultant object file has a file extension of .o by default.

The linker accepts the object file as input and clears up all external references, such as references to independently compiled external functions. Internally, segments of object code, such as the object file, are known to the linker as object modules. The object module has the same name (without an extension) as the object file by default.

The second way to compile programs is to compile several compilation units into separate object files. The linker can take more than one object file as input; then, the linker resolves references between these individual modules as well as to external references. For more information about compiling and linking, refer to Chapter 2.

8.1.2 Position of the Declaration

In determining the scope of a function or variable identifier, you must consider the position of a declaration within the program. A declaration often determines the size of a storage allocation, but a definition initiates the allocation of storage. Since declarations are often definitions, this section refers to both definitions and declarations as declarations. You may want to review Chapter 7 before reading the rest of this section.

The location of a declaration establishes the scope of an identifier. If a declaration is located inside a block, which is a segment of code delimited by braces ({}), the compiler recognizes the identifier from the point of the declaration to the end of the block. If a declaration is located outside of all functions, the compiler recognizes the identifier from the point of the declaration to the end of the compilation unit.

You can specify a storage class specifier or modifier within an identifier's declaration. A storage class specifier indicates a storage class, and a modifier modifies access to that storage. The order of the storage-class specifier, storage-class modifier, and the data-type keyword within the declaration does not matter. Consider the following example:

```
auto  int  x;    /*  And, equivalently ...   */
int  auto  x;
```

You can declare internal storage-class identifiers; the compiler recognizes these identifiers from the point of the declaration to the end of the enclosing block or function body. You can declare identifiers that are static. If the **static** declaration is outside all function bodies, the compiler recognizes these identifiers from the point of the declaration to the end of the compilation unit.

You can also declare external storage-class or global storage-class identifiers. If the declaration is outside all function bodies, the compiler recognizes these identifiers from the point of the declaration to the end of the compilation unit. The external and global storage classes differ from the static storage class in that the linker can recognize a global or external variable from the point of the declaration to the end of the program; the external and global storage classes establish a scope that can span object modules.

Table 8–1 lists the storage classes followed by the storage-class specifiers that you use to establish scope.

Table 8–1: VAX C Storage Classes and Storage-Class Specifiers

Storage Class	Specifiers
Internal	**auto**, **register**, absence of a specifier inside a block or function[1]
Static	**static**
External	**extern**, absence of a specifier outside of all functions
Global	**globaldef**, **globalref**, **globalvalue**

[1]Functions declared without a storage-class specifier are from the external storage class by default.

see the following sections for more information on storage classes:

- Section 8.3 discusses the internal storage class
- Section 8.4 discusses the static storage class
- Section 8.5 discusses the external storage class
- Section 8.6 discusses the global storage class

You can use the data-type modifiers (**const** and **volatile**) or the VAX C specific storage-class modifiers (**readonly** and **noshare**) to restrict access to data or to specify storage requirements. See Section 8.7 for more information about the data-type modifiers. See Section 8.8 for more information about the storage-class modifiers.

8.1.3 Lexical Scope and Link-Time Scope

When you use the storage-class specifiers and modifiers, be careful when positioning the definitions and declarations of your identifiers and keep the following two goals in mind to avoid error messages:

- Compile the program so that the compiler recognizes all identifiers in the compilation unit.
- Link the program so that the linker resolves all references to external data definitions.

You must make a distinction between the following types of scope:

Lexical scope — The region of a compilation unit within which an identifier is known to the compiler. In this guide, the term scope implies lexical scope.

Link-time scope	The regions of an entire program within which an external or global identifier is known to the linker. Only the identifiers in the external and global storage classes have a significant link-time scope.

Table 8–2 lists the VAX C storage-class specifiers and shows both the link-time scope and the lexical scope implied by each specifier when used inside and outside of functions.

Table 8–2: Scope and the Storage-Class Specifiers

	Inside a Function		Outside a Function	
Storage Class	**Lexical Scope**	**Link-Time Scope**	**Lexical Scope**	**Link-Time Scope**
[auto]	function	N/A	illegal	illegal
register	function	N/A	illegal	illegal
static	function	function	CU[1]	module
[extern]	function	program	CU[1]	program
globaldef	function	program	CU[1]	program
globalref	function	program	CU[1]	program
globalvalue	function	program	CU[1]	program
(null)	function	N/A	CU[1]	program

[1] Compilation Unit

The null identifier signifies the absence of a storage-class specifier from the declaration. The compiler treats a (null) inside a function or block as an identifier declared with the **[auto]** keyword; the compiler treats a (null) outside all functions as an external definition, because the identifier is from the external storage class.

In Table 8–2, the **[auto]** notation specifies identifiers of the automatic storage class. If you do not include a storage-class specifier on a definition inside of a function definition, the object is **auto** by default. This notation is used throughout this manual to represent the automatic storage class, regardless of the presence of the **[auto]** specifier in the definition. In Table 8–2, the **[extern]** notation signifies identifiers of the external storage class. A single definition exists without using a storage-class specifier; other declarations, which use the **extern** specifier, may exist which reference that definition. This notation is used throughout this chapter. See Section 8.5 for more information about the external storage class.

8.1.4 Program Example

Determining the scope of **static**, external, and global symbols can be very difficult. In Example 8–1, consider the scope of the identifiers.

The following list specifies the variable identifiers in Example 8–1, and in which functions they can be accessed without compile-time errors:

Example 8–1: Scope and Externally Defined Variables

```
Compilation Unit 1                    Compilation Unit 2
------------------                    ------------------

globaldef int  GLOBAL_1;
          int  EXT_2;                 int  EXT_1;
static    int  STAT;

f1()                                  f3()
{                                     {
   globaldef int  GLOBAL_2;              extern int  EXT_2;
      .                                     .
      .                                     .
      .                                     .
}                                     }

extern int  EXT_1;                    globalref int  GLOBAL_2;

f2()                                  f4()
{                                     {
   .                                  globalref int  GLOBAL_1;
   .                                     .
   .                                     .
}                                        .
                                      }

                                      f5()
                                      {
                                         static int  STAT;
                                         .
                                         .
                                         .
                                      }
```

Identifier	Scope
GLOBAL_1	This variable is defined outside all the functions in Compilation Unit 1, so you can access GLOBAL_1 in the functions f1 and f2 (from the point of the declaration to the end of the compilation unit). In Compilation Unit 2, the declaration of this variable is located inside function f4; the scope of the variable, in this compilation unit, only extends from the declaration of GLOBAL_1 to the end of function f4.
GLOBAL_2	This variable is defined inside function f1. In Compilation Unit 1, the scope of GLOBAL_2 only extends from the declaration of GLOBAL_2 to the end of function f1. In Compilation Unit 2, the declaration of this variable is outside all functions but is located after function f3. You can access the variable in functions f4 and f5 (from the point of the declaration to the end of the compilation unit).
EXT_1	This variable is declared outside all the functions. This declaration is a reference to the definition of the same variable in the other compilation unit. In Compilation Unit 1, you can access EXT_1 in function f2 (from the point of the declaration to the end of the compilation unit). In Compilation Unit 2, the definition of this variable is outside all the functions; you can access EXT_1 in the functions f3, f4, and f5 (from the point of the declaration to the end of the compilation unit).
EXT_2	This variable is defined outside all the functions. In Compilation Unit 1, you can access EXT_2 in the functions f1 and f2 (from the point of the declaration to the end of the compilation unit). In Compilation Unit 2, the declaration of this variable is located inside the function f3. You can access EXT_2 from the location of this declaration to the end of function f3.
STAT	There are two variables with the same name but with different permanent storage locations. In essence, these are two different variables. In Compilation Unit 1, the variable is defined outside all the functions. You can access STAT, in Compilation Unit 1, in the functions f1 and f2 (from the point of the declaration to the end of the compilation unit). In Compilation Unit 2, the separate variable is defined inside function f5; you can access STAT from this declaration to the end of function f5.

Another way to view the determination of scope is to consider the placement of the declaration as a matter of privacy. In Compilation Unit 2 in the previous example, identifier EXT_2 is made private to function f3 by placing the declaration inside the function body. If you want to keep a variable private to Compilation Unit 1, you can use a declaration using the **static** storage-class specifier. There is no way to access a variable declared with **static** in another compilation unit.

Using the **auto** and **register** internal storage-class specifiers or positioning a declarative with no storage class specifier inside a function declaration, assures privacy to the function. Internal storage is deallocated after execution of the function body. There is no way to access a variable declared with internal storage class in another function or compilation unit.

8.2 Storage Allocation

When you define a variable, the storage class determines not only its scope but also its location and lifetime. The lifetime of a variable is the length of time for which storage is allocated. Storage for a variable can be allocated in the following locations:

- On the run-time stack
- In a machine register
- In a program section (psect)

Variables that are placed on the stack or in a register are temporary. For example, variables of the **[auto]** and **register** storage class are temporary. Their lifetimes are limited to the execution of a single block or function. All declarations of the internal storage class are also definitions; the compiler generates code to establish storage at this point in the program.

Use program sections (psects), for permanent variables; the identifier's lifetimes extend through the course of the entire program. A psect represents an area of virtual memory that has a name, a size, and a series of attributes that describe the intended or permitted usage of that portion of memory. For example, the compiler places variables of the static, external, and global storage classes in psects. You have some control in determining the psects that contain identifiers. All declarations of the static storage class are also definitions; the compiler creates the psect at that point in the program. In VAX C, the first declaration of the external storage class is also a definition; the linker initializes the psect at that point in the program.

Table 8–3 shows the location and lifetime of a variable when you use each of the storage-class keywords:

Table 8–3: Location, Lifetime, and the Storage-Class Keywords

Storage Class	Location	Lifetime
[auto]	Stack or register	Temporary
register	Stack or register	Temporary
static	Psect	Permanent
[extern]	Psect	Permanent
globaldef	Psect	Permanent
globalref	Psect	Permanent
globalvalue	No storage allocated	Permanent

When working with some of the storage-class keywords, you need to know about the psects that are created by your data declarations and VAX C.

8.3 Internal Storage Class

You can assign the internal storage class to identifiers using the **auto** and **register** storage-class specifiers or by placing a declaration that contains no storage-class specifiers inside a function body. The following sections describe these specifiers.

8.3.1 The auto Specifier

Use the **auto** storage-class specifier to define a variable whose storage is allocated automatically upon entry into a function or block, and is automatically deallocated upon exit from a function or block. Within a function, **auto** is the default storage class; it is not necessary to explicitly specify it. That is, any variable (other than a function name) declared within a function without a storage-class specifier is given the **auto** storage class. Functions are of the external storage class by default. The code generated by the compiler contains instructions to allocate and deallocate **auto** storage by using machine registers and the run-time stack. Since new storage allocation occurs when you enter a block or function, you can have more than one **auto** variable with the same name as long as you declare them in separate blocks or functions. You cannot use **auto** outside of a function.

If you explicitly initialize an **auto** variable, the program code initializes the variable to that value each time the declaring block or function is activated. This initialization cannot occur if control passes into a block by some other means, such as a **goto** statement or if the block is the body of a **switch** statement. For more information about the **switch** and **goto** statements, refer to Chapter 5.

NOTE

The compiler can assign **auto** variables to machine registers, if possible. Otherwise, they are placed on the run-time stack.

Example 8–2 shows the reinitialization of two **auto** variables with the same name:

Example 8–2: Reinitializing auto Variables

```
/*  This example prints the values of two distinct auto    *
 *  variables that have the same identifier.               */

main()
{
❶   int i, x = 2;
    printf("main: %d\n",x);

    for (i = 0; i < 1; i++)
       {
❷          int x = 3;
           printf("for loop: %d\n",x);
        }

    printf("main: %d\n", x);
}
```

Key to Example 8–2:

❶ This definition of variable x extends through the entire function.

❷ This definition of variable x is limited to the **for** statement and supersedes the value of variable x in the surrounding function.

Output from Example 8–2 is as follows:

```
% example RETURN
main: 2
for loop: 3
main: 2
```

In Example 8–2, the variable x is defined twice within the main function, but the two variables do not conflict. While the **for** loop is executing, the variable x declared inside the block supersedes the variable x declared outside the block.

8.3.2 The register Specifier

Variables declared with the **register** storage class are similar to **auto** variables. You can only use the **register** internal storage class inside functions and blocks.

NOTE

The **register** storage-class specifier is the *only* specifier that you can use in a parameter declaration.

A **register** variable differs from a variable of storage class **auto** in the way that compiler-generated program code allocates storage. The **register** storage-class keyword suggests that the compiler flag the variable for placement in a machine register. This does not guarantee that the program code will place the variable in a register. The compiler checks the following conditions to determine whether or not a variable is flagged to be placed in a register:

- If the variable is not used, the optimizer may remove it entirely.
- If the program is compiled with the **–V nooptimization** option, no variables are assigned to registers. The optimization phase of the compiler determines whether a variable is a valid candidate for a register. (Optimization is enabled by default.)
- If the program contains too many register candidates, not all of them are assigned to registers.
- If the compiler detects any use of the variable that may make it inappropriate for assignment to a register, the variable is not flagged. For example, if the compiler detects the application of the address-of operator (&) to a variable that is declared with the **register** specifier, the variable is not placed in a register.

8.4 The Static Storage Class

The static storage class allows you to create permanent storage for a variable using the **static** storage-class specifier in the variable declaration. If declared inside a function, its scope begins at the declaration and spans the body of the function. If declared outside of functions, its scope is limited to the compilation unit; you cannot access a variable of the static storage class from another compilation unit.

If no initialization is present in the declaration of a variable of the static storage class, the linker initializes the variable to 0. However, unlike **auto** variables, the compiler-generated program code does not reallocate storage for a **static** variable every time control reenters a function containing the definition of a **static** variable. That is, if when exiting a function a **static** integer variable has the value of 4, the variable retains that value even if control reenters the defining function. If a **static** identifier with the same name is declared in another module,

the linker knows nothing of the other variable; the other variable has a separate psect allocation.

You can also define a function with the **static** storage class. A **static** function is not known to the linker and can be referenced only from within its defining module.

8.5 The External Storage Class

You can declare identifiers of the external storage class in the following manner:

- A definition not using another storage-class keyword, located outside all function bodies, establishes an external variable whose scope extends from the point of the definition to the end of the compilation unit.
- A declaration using the **extern** specifier, usually located in another compilation unit, is a reference to the original definition. This declaration extends the link-time scope of the variable into the second object module. If this declaration is inside a function, it extends the link-time scope from the point of the declaration to the end of the function. If this declaration is outside of a function, it extends the link-time scope from the point of the declaration to the end of the object module.
- You need not use external variable declarations (with the **extern** specifier) to see the definition of an external variable. Also, when necessary, you can use more than one **extern** declaration to reference the external definition.

Use the following rules to decide whether or not to use the **extern** specifier:

- If the variable is defined before it is referenced and the definition is in the same compilation unit, you do not need to declare the variable with the **extern** specifier.
- If the variable is defined after it is referenced, you need to first declare it with the **extern** specifier.
- If the variable is defined in a separate compilation unit, you must declare it with the **extern** specifier.

Consider the following example:

```
double D = 2.37;

main()
{
   extern int A;

   printf("a:\t%d\n", A);
   printf("d:\t%g\n", D);
}

int A = 5;
```

The main function in this program references two external variables, A and D. Since the variable D is defined before it is referenced, you do not need to declare it in the main function. Since the variable A is referenced before it is defined, it must be declared with the **extern** storage-class specifier.

In many implementations of the C language, you cannot use the **extern** specifier in a declaration that does not see an external definition elsewhere in the program. External variables occupy storage in psects of the same name as the variable identifier.

Whenever the compiler encounters the first declaration of an identifier of the external storage class in a VAX C program, it creates and initializes the psect. In VAX C, you can use the **extern** specifier in a declaration that does not see an external definition elsewhere in the program. However, this is not good programming practice, and if used, your programs might not be portable to other systems.

NOTE

In VAX C, you cannot initialize an identifier declared with the **extern** specifier.

You can specify the **noshare** modifier with external variables to create a psect with the NOSHR attribute. Similarly, you can specify the **readonly** or **const** modifier to create a psect with the NOWRT attribute. The **noshare** and **readonly** attributes are VAX C specific and are not portable.

8.6 The Global Storage Class

You can assign the global storage class to identifiers using the **globaldef**, **globalref**, or **globalvalue** storage-class specifiers. The following sections describe these specifiers.

8.6.1 The globaldef and globalref Specifiers

Use the **globaldef** specifier to define a global variable; use the **globalref** specifier to refer to a global variable defined elsewhere in the program. The specifiers **globaldef** and **globalref** are used in much the same way as with external storage class. Use **globalref** to see storage allocated elsewhere by a **globaldef** declaration.

When defining a global symbol using the **globaldef** specifier, you can place the symbol in one of three program sections: the $DATA psect (**globaldef** alone), the $CODE psect (**globaldef** with **readonly** or **const**), or a user-named psect. You can create a user-named psect by specifying its name as a string constant in braces immediately following the **globaldef** keyword, as shown in the following definition:

```
globaldef{"psect_name"}  int x = 2;
```

This definition creates a program section called psect_name and allocates the variable x in that psect. You can add any number of global variables to this psect by specifying the same psect name in other **globaldef** declarations. You can also specify the **noshare** modifier to create the psect with the NOSHR attribute. Similarly, you can specify the **readonly** or **const** modifier to create the psect with the NOWRT attribute.

You may initialize variables declared with **globaldef**. Variables declared with **globalref** may not, since these declarations see variables defined, and possibly initialized, elsewhere in the program. Initialization is possible only when storage is allocated for an object. This distinction is especially important when using the **readonly** or **const** modifier; unless the global variable is initialized when the variable is defined, its permanent value is 0.

Example 8–3 shows the use of global variables.

Example 8–3: Using Global Variables

```
/*  This example shows how global variables are used      *
 *  in VAX C programs.                                    */

❶ int ex_counter = 0;
❷ globaldef double velocity = 3.0e10;
❸ globaldef {"distance"} long miles = 100;

main()
{
   printf("   *** FIRST COMP UNIT ***\n");
   printf("counter:\t%d\n", ex_counter);
   printf("velocity:\t%g\n", velocity);
   printf("miles:\t\t%d\n\n", miles);
   fn();
   printf("   *** FIRST COMP UNIT ***\n");
   printf("counter:\t%d\n", ex_counter);
❹  printf("velocity:\t%g\n", velocity);
   printf("miles:\t\t%d\n\n", miles);
}

/*  ------------------------------------------------------  *
 *  The following code is contained in a separate         *
 *  compilation unit.                                     *
 *  ------------------------------------------------------  */

static ex_counter;
❺ globalref double velocity;
globalref long miles;

fn()
{
   ++ex_counter;
   printf("   *** SECOND COMP UNIT ***\n");
   if ( miles > 50 )
      velocity = miles * 3.1 / 200 ;
   printf("counter:\t%d\n", ex_counter);
   printf("velocity:\t%g\n", velocity);
   printf("miles:\t\t%d\n", miles);
}
```

Key to Example 8–3:

❶ The integer variable ex_counter is a variable of storage class **extern** in the first compilation unit. In the second compilation unit, a variable ex_counter is of storage class **static**. Even though they have the same identifier, the two ex_counter variables are different variables represented by two separate memory locations. The link-time scope of the second ex_counter is the module created from the second compilation unit. When control returns to the main function, the external variable ex_counter retains its original value.

❷ The variable velocity is a variable of storage class **globaldef** and is stored in the psect $DATA.

❸ The variable miles is also a variable of storage class **globaldef**, but it is stored in the user-specified psect distance.

❹ When the variable velocity prints after the function fn executes, the value will change. Global variables have only one storage location.

❺ When you reference global variables in another module, you must declare those variables in that module. In the second module, the global variables are declared with the **globalref** keyword.

Sample output from Example 8–3 is as follows:

```
% example RETURN
   *** FIRST COMP UNIT ***
counter:        0
velocity:       3.000000e+10
miles:          100
   *** SECOND COMP UNIT ***
counter:        1
velocity:       1.55
miles:          100
   *** FIRST COMP UNIT ***
counter:        0
velocity:       1.55
miles:          100
```

8.6.2 Comparing the Global and the External Storage Classes

The global storage-class specifiers define and declare objects that differ from external variables both in their storage allocation and in their correspondence to elements of other languages. Global variables provide a convenient and efficient way for a VAX C function to communicate with assembly-language programs, and with other high-level languages that support global symbol definition.

VAX C imposes no limit on the number of external variables in a single program.

NOTE

The global storage classes are VAX C specific and are not portable.

There are other differences between the external and global variables. For example:

- Global variables correspond to global symbols declared in assembly-language programs but external variables correspond to FORTRAN common blocks.
- If you have a limited amount of storage available, you may psee use the **globalvalue** specifier (see Section 8.6.3) since it does not occupy storage in your program if you can express it in 32 or fewer bits; the external variables create program sections.
- You can declare a global variable, using **globaldef**, inside a function or block and, by using a **globalref** specifier, access the identifier from another compilation unit. With external variables, you must define the variable outside all functions and blocks, and then access that variable in other compilation units by using **extern** declarations.
- A **globalref** declaration causes the linker to load the module containing the corresponding **globaldef** into the image; an **extern** declaration does not cause the linker to do so.

One similarity between the external and global storage classes is that you can place the external variables (by default) and the global variables (optionally) in psects with a user-defined name, and to some degree, user-defined attributes. The compiler places external variables in psects of the same name as the variable identifier. The compiler places **globaldef**{"name"} variables in psects with names

specified in quotation marks, delimited by braces, and located directly after the **globaldef** specifier in a declaration.

The compiler places a variable declared using only the **globaldef** specifier and a data-type keyword into the $DATA psect.

8.6.3 The globalvalue Specifier

A global value is an integral value whose identifier is a global symbol. Global values are useful because they allow many programmers in the same environment to see values by identifier, without regard to the actual value associated with the identifier. The actual values can change, as dictated by general system requirements, without requiring changes in all the programs that see them. If you make changes to the global value, you only have to recompile the defining compilation unit (unless it is defined in an object library), not all of the compilation units in the program that see those definitions.

NOTE

You can use the **globalvalue** specifier only with variables of type **enum**, **int**, or with pointer variables.

A variable declared with **globalvalue** does not require storage. Instead, the linker resolves all references to the value. If an initializer appears with **globalvalue**, the name defines a global symbol for the given initial value. If no initializer appears, the **globalvalue** construct is considered a reference to some previously defined global value.

Example 8–4 shows the use of the **globalvalue** storage-class specifier.

Example 8–4: Using the globalvalue Specifier

```
/*  This program shows references to previously defined        *
 *  globalvalue symbols.                                       */

globalvalue FAILURE = 0, EOF = -1;

main()
{
   char c;
                                    /* Get a char from stdin    */
   while ( (c = getchar())  !=  EOF)
      test(c);
}

/*  ---------------------------------------------------------  *
 *  The following code is contained in a separate compilation  *
 *  unit.                                                      *
 *  ---------------------------------------------------------  */

#include <ctype.h>                  /* Include proper module    */
globalvalue  FAILURE,  EOF;         /* Declare global symbols   */
```

(continued on next page)

Example 8–4 (Cont.): Using the globalvalue Specifier

```
test(param_c)
char param_c;                          /* Declare parameter        */
{
                                       /* Test to see if number    */
   if ( (isalnum(param_c))  !=  FAILURE)
      printf("%c\n", param_c);
   return;
}
```

In Example 8–4, FAILURE and EOF are defined in the first module: the values are placed in the program stream. In the second module, FAILURE and EOF are declared so that their values can be accessed.

8.6.4 Global Enumerated Types

When you use the **globaldef** storage-class keyword with an **enum** definition, the enumerated constants in the definition are of the **globalvalue** storage class, and initialized as required by the program to form a properly ordered list of the values. Enumerated type variables are of the **globaldef** storage class.

When you use **globalref** with the **enum** keyword, all enumerated variables are of the storage class **globalref**, and the enumerated constants see global values of the same names as shown in the following example.

The first compilation unit includes the following sequence:

```
globaldef enum  light { dim, medium = 3, bright }  light_val;

main()

{
    light_val = dim;
    fnlv();
}
```

The second compilation unit includes the following sequence:

```
globalref enum  light { dim, medium, bright }  light_val;

fnlv()
{
    if (light_val < bright ) printf("TOO DIM\n");
}
```

In the first compilation unit, the **enum** definition establishes light_val as a **globaldef** of the enumerated type light. It also establishes the ordered list of enumerated global values dim, medium, and bright.

The **globalref** declaration in the second compilation unit allows the enumerated constants to be used as global values. That is, the constants can be referenced, but they cannot be initialized.

For more information about the enumerated type, see Chapter 7.

8.7 Data-type modifiers

Data-type modifiers affect the allocation or access of data storage. The data-type modifiers are as follows:

- **const**
- **volatile**

The following sections describe these data-type modifiers in detail.

8.7.1 The const Modifier

The **const** data-type modifier restricts access to stored data. If you declare an object to be of type **const**, you cannot modify that object.

The following rules apply to the use of the **const** data-type modifier:

- You can specify **const** with any of the other data-type keywords in a declaration.
- If you specify **const** when declaring an aggregate, all the aggregate members are treated as objects of type **const**.
- You can specify **const** with **volatile**, or any of the storage-class specifiers or modifiers.
- If you try to access a **const** object using a pointer to an object not declared as **const**, the result is undefined.
- The address of a non-**const** object can be assigned to a pointer to a **const** object, but you cannot use that pointer to alter the value of the object. The result is undefined.

The following example declares the variable x to be a constant integer.

```
int const x;
```

When declaring pointers, depending upon the placement of the **const** modifier in the declaration, VAX C will either interpret the pointer or the object to which it points as the constant variable. For example, the following section of code declares the variable y to be a constant pointer to an integer because the **const** modifier appears after the asterisk:

```
int * const y;
```

In the next example, the variable z is declared as a pointer to a constant integer because the asterisk appears after the **const** modifier:

```
int const * z;
```

When you specify the **const** modifier in association with a **globaldef** specifier that identifies a psect, be aware that all variables declared have their storage allocated in the psect and that an inconsistent use of the **const** modifier can alter the psect attribute and lead to diagnostic messages. Examples 1 and 2 show invalid uses of the **const** modifiers. Specifically, in Example 1 the variable x becomes a nonconstant pointer to a constant integer and therefore assigns the WRT attribute to the psect. In Example 2, the variable y becomes a constant pointer to an integer and assigns the NOWRT attribute to the psect. In Example 3, the variable z becomes a constant variable contained in the psect and assigns it the NOWRT attribute.

Example 1

```
globaldef {"psect"} const int * x;   /* invalid example  */
```

Example 2

```
globaldef {"psect"} int * const y;   /* invalid example  */
```

Example 3

```
globaldef {"psect"} const int z;
```

VAX C generates a warning message when there is an inconsistent usage of the **const** modifier, as shown in the following example:

```
globaldef {"psect"} const int test, * bar;
```

In this example, the variable test is declared as a constant variable that becomes allocated in the psect and assigns it the NOWRT attribute. The variable bar is a pointer that is not itself constant, but that points to a constant integer. In this case, VAX C automatically causes the pointer to become constant. Therefore, Digital recommends that you do not mix constant and nonconstant variables in a **globaldef** declaration that names a psect, or your program may generate unpredictable results.

8.7.2 The volatile Modifier

The **volatile** data-type modifier prevents an object from being stored in a machine register, forcing it to be allocated in memory. This data-type modifier is useful for declaring data that is to be accessed asynchronously. A device driver application often uses **volatile** data storage.

The following rules apply to the use of the **volatile** modifier:

- You can specify **volatile** with any of the other data-type keywords in a declaration.
- If you specify **volatile** when declaring an aggregate, all of the aggregate members are treated as objects of type **volatile**.
- You can specify **volatile** with **const**, or any of the storage-class specifiers or modifiers except the storage class **register**.
- The address of an object of some other type can be assigned to a **volatile** pointer, but the rules of the **volatile** data-type modifier must be followed if you see the object using that pointer.

8.8 Storage-Class Modifiers

The VAX C compiler can accept a storage-class specifier and a storage-class modifier in any order; usually, the modifier is placed after the specifier in the source code. For example:

```
extern  noshare  int  x;
   /*  Or, equivalently...  */
int  noshare  extern  x;
```

The following sections describe each of the VAX C specific storage class modifiers in detail.

8.8.1 The noshare Modifier

The **noshare** storage-class modifier assigns the attribute NOSHR to the program section of the variable. This storage-class modifier is relevant only when used with VMS shared images. It is retained in VAX C/ULTRIX for reasons of compatibility with VAX C/VMS and has no meaning in VAX C/ULTRIX.

The **noshare** modifier can be used with the storage-class specifiers **static**, **[extern]**, **globaldef**, and **globaldef**{"name"}.

You can use **noshare** alone; when you do this, an external definition of storage class **[extern]** is implied. Also, when declaring variables using the **[extern]** and **globaldef**{"name"} storage-class specifiers, you can use **noshare**, **const**, and **readonly**, together, in the declaration. If you declare variables using the **static** or the **globaldef** specifiers, and you use both of the modifiers in the declaration, the compiler ignores **noshare** and accepts **const** or **readonly**.

8.8.2 The readonly Modifier

The **readonly** storage-class modifier, like the **const** data-type modifier, assigns the NOWRT attribute to the variable's program section; if used with the **static** or **globaldef** specifier, the variable is stored in the psect $CODE, which has the NOWRT attribute by default.

Both the **readonly** and **const** modifiers can be used with the storage-class specifiers **static**, **[extern]**, **globaldef**, and **globaldef** {"psect"}.

In addition, both the **readonly** modifier and the **const** modifier can be used alone. When you specify these modifiers alone, an external definition of storage class **[extern]** is implied.

The **readonly** modifier restricts access to data in the same manner as the **const** modifier. However, in the declaration of a pointer, the **readonly** modifier cannot appear between the asterisk and the pointer variable to which it applies.

The following example shows the similarity between the **const** and **readonly** modifiers. In both instances, the variable point represents a constant pointer to a nonconstant integer.

```
readonly int * point;
int * const point;
```

NOTE

For new program development, Digital recommends that you use the **const** modifier.

8.8.3 The _align Modifier

The **_align** modifier allows you to align objects of any of the VAX C data types on a specified storage boundary. You use the **_align** modifier in a data declaration or definition.

For example, if you want to align an integer on the next quadword boundary, you can use any of the following declarations:

```
int  _align( QUADWORD )  data;
int  _align( quadword )  data;
int  _align( 3 )  data;
```

When specifying the boundary of the data alignment, you can either use a predefined constant or you can specify an integer value that is a power of two. The power of two tells VAX C the number of bytes to pad in order to align the data. So, in the previous example, integer 3 specifies an alignment of 2^3 bytes, which is 8 bytes—a quadword of memory.

The following list presents all of the predefined alignment constants, their equivalent power of two, and their equivalent number of bytes.

Constant	Power of Two	Number of Bytes
BYTE or byte	0	0
WORD or word	1	2
LONGWORD or longword	2	4
QUADWORD or quadword	3	8
OCTAWORD or octaword	4	16
PAGE or page	9	512